

PILOT-03: Cost Agent Skeleton (FastAPI)

CONTEXT

Phase: PILOT (Week 0)

Component: Cost Agent - Basic FastAPI Application



Estimated Time: 25 min AI execution + 15 min verification

Complexity: Medium

Risk Level: MEDIUM (tests FastAPI generation, agent registration)

DEPENDENCIES

Must Complete First:

- **PILOT-01:** Bootstrap project structure  COMPLETED
- **PILOT-02:** Orchestrator skeleton  COMPLETED

Required Services Running:

```
bash

# Verify infrastructure
make verify
# Expected: PostgreSQL, ClickHouse, Qdrant, Redis - all HEALTHY

# Verify orchestrator
curl http://localhost:8080/health
# Expected: {"status":"healthy",...}
```

Required Environment:

```
bash

# Python installed
python --version # Python 3.11+

# Project structure exists
ls services/cost-agent/
```

OBJECTIVE

Create a **Python FastAPI application** that will serve as the Cost Agent. This agent will eventually optimize cloud costs through spot migrations, right-sizing, and reserved instances.

Success Criteria:

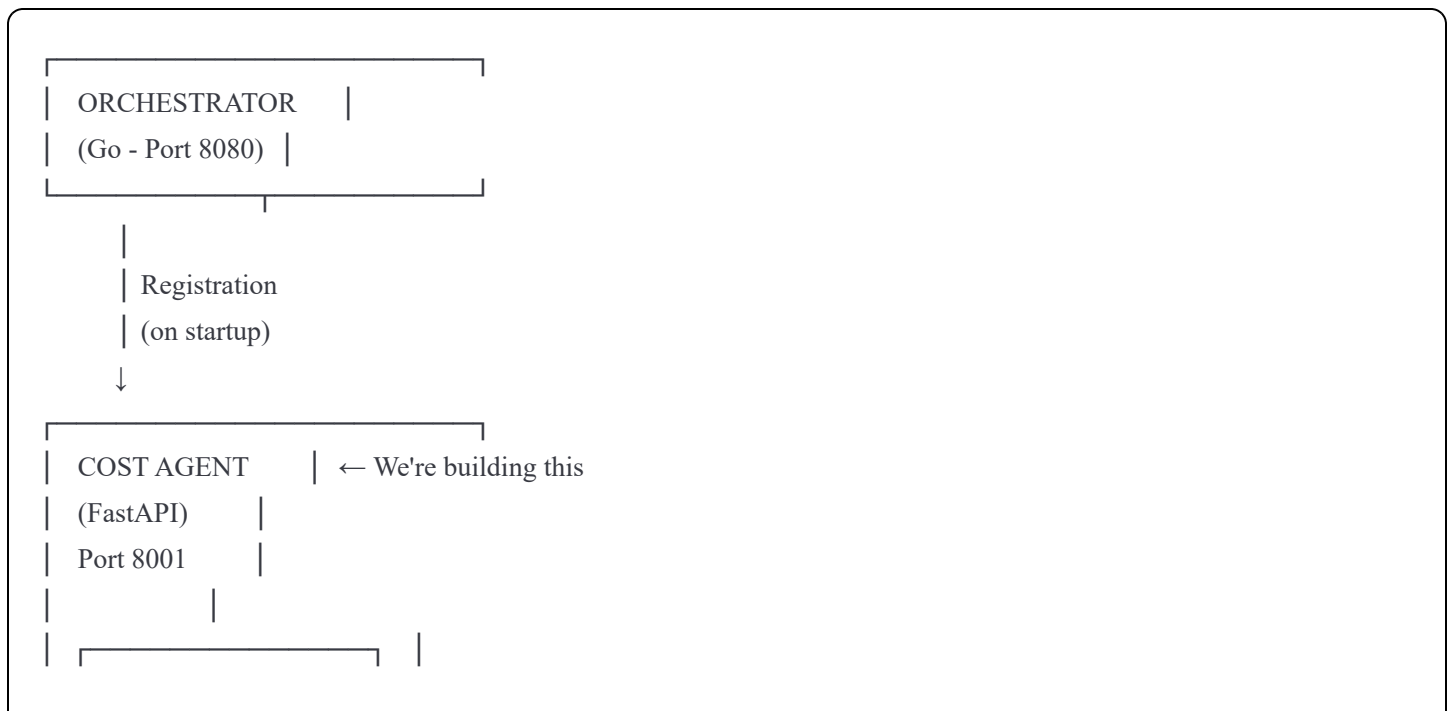
- ✓ FastAPI application starts on port 8001
- ✓ `/health` endpoint returns 200 OK
- ✓ Agent registers with orchestrator automatically
- ✓ Structured logging works (JSON format)
- ✓ Configuration loads from environment
- ✓ Docker image builds successfully (< 200 MB)
- ✓ Basic tests pass (80%+ coverage)

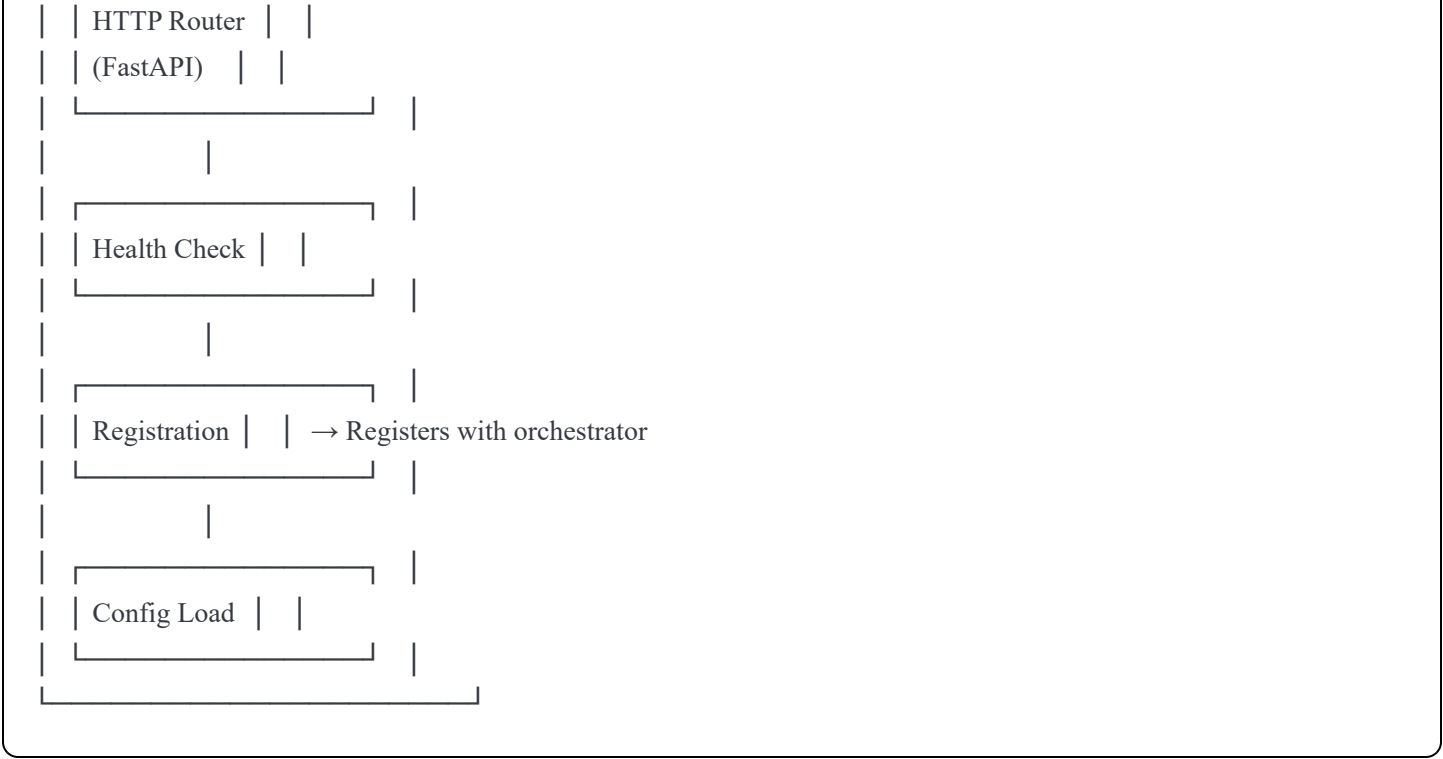
Failure Signs:

- ✗ FastAPI app won't start
- ✗ Registration with orchestrator fails
- ✗ Health endpoint returns errors
- ✗ Docker build fails
- ✗ Import errors or missing dependencies

TECHNICAL SPECIFICATION

Architecture Context





File Structure to Create

```
services/cost-agent/
├── src/
│   ├── __init__.py      # CREATE: Package init
│   ├── main.py          # CREATE: FastAPI app entry
│   ├── config.py        # CREATE: Configuration
│   └── models/
│       ├── __init__.py
│       └── health.py    # CREATE: Health response model
│   └── api/
│       ├── __init__.py
│       └── health.py    # CREATE: Health endpoint
│   └── core/
│       ├── __init__.py
│       ├── logger.py    # CREATE: Structured logging
│       └── registration.py # CREATE: Orchestrator registration
├── tests/
│   ├── __init__.py
│   ├── conftest.py      # CREATE: Pytest fixtures
│   └── test_health.py   # CREATE: Health endpoint tests
├── requirements.txt     # CREATE: Python dependencies
├── Dockerfile           # CREATE: Multi-stage build
└── .dockerignore        # CREATE: Docker ignore
```

└─ pyproject.toml	# CREATE: Project metadata
└─ pytest.ini	# CREATE: Pytest configuration
└─ README.md	# CREATE: Documentation

IMPLEMENTATION REQUIREMENTS

1. requirements.txt

```
txt

# FastAPI and server
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
pydantic-settings==2.1.0

# HTTP client for orchestrator registration
httpx==0.25.1

# Database (for later use)
sqlalchemy==2.0.23
asyncpg==0.29.0

# Logging
python-json-logger==2.0.7

# Testing
pytest==7.4.3
pytest-asyncio==0.21.1
pytest-cov==4.1.0
httpx==0.25.1

# Development
black==23.11.0
flake8==6.1.0
mypy==1.7.1
```

2. src/main.py (FastAPI Application)

```
python
```

```
"""
```

OptiInfra Cost Agent - Main Application

This is the Cost Agent that optimizes cloud spending through:

- Spot instance migrations
- Reserved instance recommendations
- Instance right-sizing

```
"""
```

```
import asyncio
```

```
import logging
```

```
from contextlib import asynccontextmanager
```

```
from fastapi import FastAPI
```

```
from fastapi.middleware.cors import CORSMiddleware
```

```
from src.api import health
```

```
from src.config import settings
```

```
from src.core.logger import setup_logging
```

```
from src.core.registration import register_with_orchestrator
```

```
# Setup logging
```

```
logger = setup_logging()
```

```
@asynccontextmanager
```

```
async def lifespan(app: FastAPI):
```

```
    """
```

```
    Lifespan events for the FastAPI application.
```

```
    Handles startup and shutdown events.
```

```
    """
```

```
# Startup
```

```
logger.info("Starting OptiInfra Cost Agent")
```

```
logger.info(f"Environment: {settings.environment}")
```

```
logger.info(f"Port: {settings.port}")
```

```
# Register with orchestrator
```

```
if settings.orchestrator_url:
```

```
    try:
```

```
        await register_with_orchestrator()
```

```
        logger.info("Successfully registered with orchestrator")
```

```
    except Exception as e:
```

```
        logger.error(f"Failed to register with orchestrator: {e}")
```

```
# Don't fail startup if registration fails
```

yield

Shutdown

logger.info("Shutting down Cost Agent")

Create FastAPI app

```
app = FastAPI(
    title="OptiInfra Cost Agent",
    description="AI-powered cost optimization agent",
    version="0.1.0",
    lifespan=lifespan,
)
```

Add CORS middleware

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # Configure based on environment
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Include routers

```
app.include_router(health.router, tags=["health"])
```

Root endpoint

```
@app.get("/")
async def root():
    """Root endpoint - service information"""
    return {
        "service": "OptiInfra Cost Agent",
        "version": "0.1.0",
        "status": "running",
        "capabilities": [
            "spot_migration",
            "reserved_instances",
            "right_sizing",
        ],
    }
```

Run with uvicorn when executed directly

```
if __name__ == "__main__":  
    import uvicorn  
  
    uvicorn.run(  
        "src.main:app",  
        host="0.0.0.0",  
        port=settings.port,  
        reload=settings.environment == "development",  
        log_level=settings.log_level.lower(),  
    )
```

3. src/config.py (Configuration)

python

```

"""
Configuration management for Cost Agent.
Loads settings from environment variables.
"""

from typing import Optional

from pydantic_settings import BaseSettings


class Settings(BaseSettings):
    """Application settings loaded from environment variables"""

    # Application
    app_name: str = "OptiInfra Cost Agent"
    environment: str = "development"
    port: int = 8001
    log_level: str = "INFO"

    # Orchestrator
    orchestrator_url: Optional[str] = "http://localhost:8080"
    agent_id: str = "cost-agent-001"
    agent_type: str = "cost"

    # Database (for later use)
    database_url: Optional[str] = None

    # Redis (for later use)
    redis_url: Optional[str] = None

    class Config:
        env_file = ".env"
        case_sensitive = False

    # Global settings instance
    settings = Settings()

```

4. src/models/health.py (Pydantic Models)

```
python
```

```
"""
```

Health check response models.

```
"""
```

```
from datetime import datetime
```

```
from typing import Optional
```

```
from pydantic import BaseModel, Field
```

```
class HealthResponse(BaseModel):
```

```
    """Health check response model"""
```

```
    status: str = Field(..., description="Health status")
```

```
    timestamp: datetime = Field(default_factory=datetime.utcnow)
```

```
    version: str = Field(..., description="Application version")
```

```
    agent_id: str = Field(..., description="Agent identifier")
```

```
    agent_type: str = Field(..., description="Agent type")
```

```
    uptime_seconds: float = Field(..., description="Uptime in seconds")
```

```
class Config:
```

```
    json_schema_extra = {
```

```
        "example": {
```

```
            "status": "healthy",
```

```
            "timestamp": "2025-10-17T10:00:00Z",
```

```
            "version": "0.1.0",
```

```
            "agent_id": "cost-agent-001",
```

```
            "agent_type": "cost",
```

```
            "uptime_seconds": 120.5,
```

```
        }
```

```
    }
```

```
class AgentRegistration(BaseModel):
```

```
    """Agent registration model for orchestrator"""
```

```
    agent_id: str = Field(..., description="Unique agent identifier")
```

```
    agent_type: str = Field(..., description="Type of agent")
```

```
    host: str = Field(..., description="Agent host")
```

```
    port: int = Field(..., description="Agent port")
```

```
    capabilities: list[str] = Field(..., description="Agent capabilities")
```

```
class Config:
```

```
    json_schema_extra = {
```

```
"example": {  
  "agent_id": "cost-agent-001",  
  "agent_type": "cost",  
  "host": "localhost",  
  "port": 8001,  
  "capabilities": ["spot_migration", "reserved_instances", "right_sizing"],  
}
```

5. src/api/health.py (Health Endpoint)

```
python
```

```

"""
Health check endpoint for Cost Agent.
"""

import time

from fastapi import APIRouter

from src.config import settings
from src.models.health import HealthResponse

router = APIRouter()

# Track startup time
_start_time = time.time()

@router.get("/health", response_model=HealthResponse)
async def health_check():
    """
    Health check endpoint.

    Returns:
        HealthResponse: Current health status with uptime
    """
    uptime = time.time() - _start_time

    return HealthResponse(
        status="healthy",
        version="0.1.0",
        agent_id=settings.agent_id,
        agent_type=settings.agent_type,
        uptime_seconds=uptime,
    )

```

6. src/core/logger.py (Structured Logging)

```
python
```

```

"""
Structured logging setup for Cost Agent.
"""

import logging
import sys

from pythonjsonlogger import jsonlogger

from src.config import settings


def setup_logging() -> logging.Logger:
    """
    Setup structured JSON logging.

    Returns:
        logging.Logger: Configured logger instance
    """
    logger = logging.getLogger("cost_agent")

    # Set log level
    log_level = getattr(logging, settings.log_level.upper(), logging.INFO)
    logger.setLevel(log_level)

    # Create handler
    handler = logging.StreamHandler(sys.stdout)
    handler.setLevel(log_level)

    # Create JSON formatter
    formatter = jsonlogger.JsonFormatter(
        fmt="%(asctime)s %(levelname)s %(name)s %(message)s",
        datefmt="%Y-%m-%dT%H:%M:%SZ",
    )
    handler.setFormatter(formatter)

    # Add handler to logger
    logger.addHandler(handler)

    return logger

```

7. src/core/registration.py (Orchestrator Registration)

```
python
```

```
"""
```

Agent registration with orchestrator.

```
"""
```

```
import logging
```

```
import httpx
```

```
from src.config import settings
```

```
from src.models.health import AgentRegistration
```

```
logger = logging.getLogger("cost_agent")
```

```
async def register_with_orchestrator() -> None:
```

```
    """
```

Register this agent with the orchestrator.

Sends agent information to orchestrator so it can route requests.

```
    """
```

```
    if not settings.orchestrator_url:
```

```
        logger.warning("Orchestrator URL not configured, skipping registration")
```

```
        return
```

```
    registration_data = AgentRegistration(
```

```
        agent_id=settings.agent_id,
```

```
        agent_type=settings.agent_type,
```

```
        host="cost-agent", # Docker service name
```

```
        port=settings.port,
```

```
        capabilities=[
```

```
            "spot_migration",
```

```
            "reserved_instances",
```

```
            "right_sizing",
```

```
        ],
```

```
    )
```

```
    async with httpx.AsyncClient() as client:
```

```
        try:
```

```
            # Note: This endpoint doesn't exist yet (will be added in 0.6)
```

```
            # For now, just log that we would register
```

```
            logger.info(
```

```
                f'Would register with orchestrator at {settings.orchestrator_url}/agents/register'
```

```
            )
```

```
            logger.info(f'Registration data: {registration_data.model_dump()}')
```

```

# Uncomment when orchestrator has /agents/register endpoint (Prompt 0.6)
# response = await client.post(
#     f"{settings.orchestrator_url}/agents/register",
#     json=registration_data.model_dump(),
#     timeout=5.0,
# )
# response.raise_for_status()
# logger.info("Successfully registered with orchestrator")

except httpx.HTTPError as e:
    logger.error(f"Failed to register with orchestrator: {e}")
    raise

```

8. tests/conftest.py (Pytest Fixtures)

```

python

"""
Pytest configuration and fixtures.
"""

import pytest
from fastapi.testclient import TestClient

from src.main import app

@pytest.fixture
def client():
    """
    Test client for FastAPI app.
    """
    return TestClient(app)

@pytest.fixture
def mock_settings(monkeypatch):
    """
    Mock settings for testing.
    """
    monkeypatch.setenv("ORCHESTRATOR_URL", "http://localhost:8080")
    monkeypatch.setenv("AGENT_ID", "test-agent-001")
    monkeypatch.setenv("ENVIRONMENT", "test")

```

9. tests/test_health.py (Health Endpoint Tests)

python

```
"""
```

Tests for health endpoint.

```
"""
```

```
import pytest
```

```
from fastapi.testclient import TestClient
```

```
def test_health_endpoint_returns_200(client: TestClient):
```

```
    """Test that health endpoint returns 200 OK"""
```

```
    response = client.get("/health")
```

```
    assert response.status_code == 200
```

```
def test_health_endpoint_has_correct_structure(client: TestClient):
```

```
    """Test that health response has correct structure"""
```

```
    response = client.get("/health")
```

```
    data = response.json()
```

```
    assert "status" in data
```

```
    assert "timestamp" in data
```

```
    assert "version" in data
```

```
    assert "agent_id" in data
```

```
    assert "agent_type" in data
```

```
    assert "uptime_seconds" in data
```

```
def test_health_status_is_healthy(client: TestClient):
```

```
    """Test that health status is 'healthy'"""
```

```
    response = client.get("/health")
```

```
    data = response.json()
```

```
    assert data["status"] == "healthy"
```

```
def test_health_agent_type_is_cost(client: TestClient):
```

```
    """Test that agent type is 'cost'"""
```

```
    response = client.get("/health")
```

```
    data = response.json()
```

```
    assert data["agent_type"] == "cost"
```

```
def test_health_version_is_present(client: TestClient):
```

```

"""Test that version is present"""
response = client.get("/health")
data = response.json()

assert data["version"] == "0.1.0"


def test_root_endpoint_returns_200(client: TestClient):
    """Test that root endpoint returns 200 OK"""
    response = client.get("/")
    assert response.status_code == 200


def test_root_endpoint_has_capabilities(client: TestClient):
    """Test that root endpoint lists capabilities"""
    response = client.get("/")
    data = response.json()

    assert "capabilities" in data
    assert "spot_migration" in data["capabilities"]
    assert "reserved_instances" in data["capabilities"]
    assert "right_sizing" in data["capabilities"]


def test_health_uptime_increases(client: TestClient):
    """Test that uptime increases between calls"""
    import time

    response1 = client.get("/health")
    time.sleep(0.1)
    response2 = client.get("/health")

    uptime1 = response1.json()["uptime_seconds"]
    uptime2 = response2.json()["uptime_seconds"]

    assert uptime2 > uptime1

```

10. Dockerfile

dockerfile

Build stage

FROM python:3.11-slim **as** builder

WORKDIR /build

Install build dependencies

RUN apt-get update && apt-get install -y --no-install-recommends \
gcc \
&& rm -rf /var/lib/apt/lists/*

Copy requirements

COPY requirements.txt .

Install Python dependencies

RUN pip install --no-cache-dir --user -r requirements.txt

Runtime stage

FROM python:3.11-slim

WORKDIR /app

Copy Python dependencies from builder

COPY --from=builder /root/.local /root/.local

Copy application code

COPY src/ ./src/

COPY pyproject.toml .

Make sure scripts in .local are usable

ENV PATH=/root/.local/bin:\$PATH

Expose port

EXPOSE 8001

Health check

HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
CMD python -c "import httpx; httpx.get('http://localhost:8001/health').raise_for_status()" || exit 1

Run FastAPI with uvicorn

CMD ["python", "-m", "uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "8001"]

11. .dockerignore

```
# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
*.egg-info/
.installed.cfg
*.egg

# Testing
.pytest_cache/
.coverage
htmlcov/
.tox/

# Virtual environments
venv/
env/
ENV/
.venv

# IDE
.vscode/
.idea/
*.swp
*.swo

# Misc
```

```
.DS_Store  
*.log  
.env
```

12. pyproject.toml

```
toml  
  
[tool.black]  
line-length = 88  
target-version = ['py311']  
include = '\.pyi?$'  
  
[tool.pytest.ini_options]  
testpaths = ["tests"]  
python_files = "test_*.py"  
python_classes = "Test*"  
python_functions = "test_*"  
addopts = "-v --cov=src --cov-report=html --cov-report=term-missing"  
  
[tool.mypy]  
python_version = "3.11"  
warn_return_any = true  
warn_unused_configs = true  
disallow_untyped_defs = true
```

13. pytest.ini

```
ini  
  
[pytest]  
testpaths = tests  
python_files = test_*.py  
python_classes = Test*  
python_functions = test_*
```

14. README.md

```
markdown
```

OptiInfra Cost Agent

Python FastAPI-based agent for cloud cost optimization.

Features

- FastAPI web framework
- Structured JSON logging
- Health check endpoint
- Automatic orchestrator registration
- Docker support
- Comprehensive tests (80%+ coverage)

Capabilities

- **Spot Migration**: Migrate on-demand instances to spot instances (30-40% savings)
- **Reserved Instances**: Recommend RI purchases (40-60% savings)
- **Right-Sizing**: Identify over-provisioned instances (20-30% savings)

Development

Prerequisites

- Python 3.11+
- Docker (optional)

Running Locally

```
``bash
# Create virtual environment
python -m venv venv
source venv/bin/activate # On Windows: venv\Scripts\activate

# Install dependencies
pip install -r requirements.txt

# Run
python src/main.py

# Or with uvicorn
uvicorn src.main:app --reload --port 8001
``
```

Testing

```
```bash
Run tests
pytest

With coverage
pytest --cov=src --cov-report=html

View coverage
open htmlcov/index.html
```
```

Code Quality

```
```bash
Format code
black src/ tests/

Lint
flake8 src/ tests/

Type check
mypy src/
```
```

API Endpoints

GET /health

Health check endpoint.

****Response:****

```
```json
{
 "status": "healthy",
 "timestamp": "2025-10-17T10:00:00Z",
 "version": "0.1.0",
 "agent_id": "cost-agent-001",
 "agent_type": "cost",
 "uptime_seconds": 120.5
}
```
```

GET /

Service information endpoint.

****Response:****

```
```json
{
 "service": "OptiInfra Cost Agent",
 "version": "0.1.0",
 "status": "running",
 "capabilities": [
 "spot_migration",
 "reserved_instances",
 "right_sizing"
]
}
```
```

Configuration

Environment variables:

- **`PORT`** - Port to listen on (default: 8001)
- **`ENVIRONMENT`** - Environment name (default: development)
- **`LOG_LEVEL`** - Log level (default: INFO)
- **`ORCHESTRATOR_URL`** - Orchestrator URL (default: http://localhost:8080)
- **`AGENT_ID`** - Agent identifier (default: cost-agent-001)

Docker

```
```bash
Build image
docker build -t optiinfra-cost-agent .

Run container
docker run -p 8001:8001 optiinfra-cost-agent

Or use docker-compose (from project root)
docker-compose up cost-agent
```
```

Next Steps

After this pilot phase:

- Add cloud provider collectors (1.2-1.4)
- Add LangGraph workflows (1.5-1.6)
- Add analysis engine (1.7)

- Add LLM integration (1.8)
- Add execution engine (1.10)

✓ VALIDATION COMMANDS

Step 1: Verify Python Environment

```
bash

cd services/cost-agent

# Check Python version
python --version
# Expected: Python 3.11.x or higher

# Create virtual environment
python -m venv venv

# Activate virtual environment
# On Windows:
venv\Scripts\activate
# On macOS/Linux:
source venv/bin/activate

# Verify activation
which python # Should show path in venv/
```

Step 2: Install Dependencies

```
bash

# Install requirements
pip install -r requirements.txt

# Expected output:
# Successfully installed fastapi-0.104.1 uvicorn-0.24.0 ...

# Verify installation
pip list | grep fastapi
# Expected: fastapi 0.104.1
```

Step 3: Run Tests

```
bash
```

```
# Run all tests
```

```
pytest
```

```
# Expected output:
```

```
# ===== 8 passed in 0.50s =====
```

```
# Run with coverage
```

```
pytest --cov=src --cov-report=term-missing
```

```
# Expected: Coverage >80%
```

Step 4: Start FastAPI Server

```
bash
```

```
# Start server
```

```
python src/main.py
```

```
# Expected output:
```

```
# {"asctime": "2025-10-17T10:00:00Z", "levelname": "INFO", "name": "cost_agent", "message": "Starting OptiInfra Cost Agent"}
```

```
# {"asctime": "2025-10-17T10:00:00Z", "levelname": "INFO", "name": "cost_agent", "message": "Would register with orchestra"}
```

```
# INFO:   Uvicorn running on http://0.0.0.0:8001
```

Step 5: Test Health Endpoint

In another terminal:

```
bash
```

Test health endpoint

curl http://localhost:8001/health

Expected output:

```
# {  
#   "status": "healthy",  
#   "timestamp": "2025-10-17T10:00:00Z",  
#   "version": "0.1.0",  
#   "agent_id": "cost-agent-001",  
#   "agent_type": "cost",  
#   "uptime_seconds": 5.2  
# }
```

Test root endpoint

curl http://localhost:8001/

Expected output:

```
# {  
#   "service": "OptiInfra Cost Agent",  
#   "version": "0.1.0",  
#   "status": "running",  
#   "capabilities": ["spot_migration", "reserved_instances", "right_sizing"]  
# }
```

Check response code

curl -I http://localhost:8001/health

Expected: HTTP/1.1 200 OK

Step 6: Test Auto-Documentation

bash

Open API docs

In browser: http://localhost:8001/docs

Or test with curl

curl http://localhost:8001/docs

Expected: HTML page with Swagger UI

Alternative docs

curl http://localhost:8001/redoc

Expected: HTML page with ReDoc UI

Step 7: Build Docker Image

```
bash

# Stop local server first (Ctrl+C)

# Build Docker image
docker build -t optiinfra-cost-agent:latest .

# Expected output:
# Successfully built [image-id]
# Successfully tagged optiinfra-cost-agent:latest

# Check image size
docker images optiinfra-cost-agent
# Expected: < 200 MB
```

Step 8: Run in Docker

```
bash

# Run container
docker run -d -p 8001:8001 --name cost-agent optiinfra-cost-agent:latest

# Check container is running
docker ps | grep cost-agent
# Expected: Container running

# Check logs
docker logs cost-agent
# Expected: Startup logs, no errors

# Test health endpoint
curl http://localhost:8001/health
# Expected: {"status":"healthy",...}

# Stop container
docker stop cost-agent
docker rm cost-agent
```

Step 9: Test Orchestrator Connection

```
bash
```

Make sure orchestrator is running

curl http://localhost:8080/health

Expected: {"status": "healthy", ...}

Start cost agent (it will try to register)

python src/main.py

Check logs for registration attempt

Expected: "Would register with orchestrator at http://localhost:8080/agents/register"

Step 10: Run Code Quality Checks

bash

Format code

black src/ tests/

Expected: All done! ✨

Lint

flake8 src/ tests/

Expected: No errors

Type check (optional)

mypy src/

Expected: Success: no issues found



SUCCESS CRITERIA CHECKLIST

After running all validation commands, verify:

- ☐ Python 3.11+ installed
- ☐ Virtual environment created and activated
- ☐ All dependencies installed (fastapi, uvicorn, etc.)
- ☐ All 8 tests pass
- ☐ Test coverage > 80%
- ☐ Server starts on port 8001
- ☐ `/health` endpoint returns 200 OK
- ☐ Health response has correct structure
- ☐ `/` endpoint returns service info
- ☐ Swagger docs accessible at `/docs`

- ☐ Registration attempt logged (orchestrator endpoint doesn't exist yet)
- ☐ Structured logging outputs JSON
- ☐ Docker image builds successfully
- ☐ Docker image size < 200 MB
- ☐ Container runs without errors
- ☐ Health check passes in Docker
- ☐ Code formatted (black)
- ☐ No linting errors (flake8)

Expected Time: < 40 minutes total (25 min generation + 15 min verification)

TROUBLESHOOTING

Issue 1: Python version too old

Solution:

```
bash

# Install Python 3.11+
# On Windows: Download from python.org
# On macOS: brew install python@3.11
# On Linux: apt install python3.11

# Use specific version
python3.11 -m venv venv
```

Issue 2: pip install fails

Solution:

```
bash

# Upgrade pip
python -m pip install --upgrade pip

# Install with verbose output
pip install -v -r requirements.txt

# If SSL errors, try:
pip install --trusted-host pypi.org --trusted-host files.pythonhosted.org -r requirements.txt
```

Issue 3: Server won't start - "port already in use"

Solution:

```
bash

# Check what's using port 8001
# On Windows:
netstat -ano | findstr :8001
# On macOS/Linux:
lsof -i :8001

# Kill the process or change port
export PORT=8002
python src/main.py
```

Issue 4: Tests fail with import errors

Solution:

```
bash

# Make sure you're in the virtual environment
which python # Should show venv path

# Reinstall in editable mode
pip install -e .

# Or add src to PYTHONPATH
export PYTHONPATH="${PYTHONPATH}:${pwd}"
```

Issue 5: Docker build fails

Solution:

```
bash
```

Check Docker daemon

`docker info`

Try with --no-cache

`docker build --no-cache -t optiinfra-cost-agent .`

Check for syntax errors

`docker build --progress=plain -t optiinfra-cost-agent .`

Issue 6: Health check returns 404

Solution:

`bash`

Verify server is running

`curl http://localhost:8001/`

Should return service info

Check logs for errors

Look for startup messages

Try explicit localhost

`curl http://127.0.0.1:8001/health`

DELIVERABLES

This prompt should generate:

1. Python Source Files (7 files):

- `src/main.py` (FastAPI app)
- `src/config.py` (settings)
- `src/models/health.py` (Pydantic models)
- `src/api/health.py` (health endpoint)
- `src/core/logger.py` (logging)
- `src/core/registration.py` (orchestrator registration)

2. Test Files (2 files):

- `tests/conftest.py` (fixtures)

- tests/test_health.py (8+ tests)

3. Configuration Files:

- requirements.txt
- pyproject.toml
- pytest.ini

4. Docker Files:

- Dockerfile
- .dockerignore

5. Documentation:




- README.md

6. Working FastAPI Application:

- Starts on port 8001
- Health check endpoint
- Auto-registration attempt
- Structured logging
- 80%+ test coverage

NEXT STEPS

After this prompt succeeds:

1.  **Verify:** Server running, health check works, tests pass
2.  **Commit:** `git add . && git commit -m "PILOT-03: Cost Agent skeleton (FastAPI)"`
3.  **Continue:** PILOT-04 (LangGraph Setup) - **CRITICAL**

What we'll add later:

- Cloud collectors (AWS/GCP/Azure) - Week 2
- LangGraph workflows - PILOT-04 (next)
- Analysis engine - Week 2
- LLM integration - Week 2

- Execution engine - Week 2
-

NOTES FOR WINDSURF

IMPORTANT INSTRUCTIONS:

1. **Use correct Python conventions** - PEP 8 formatting
2. **Generate complete files** - No "TODO" or placeholders
3. **Use async/await** - FastAPI is async
4. **Proper Pydantic models** - Use v2 syntax
5. **Structured logging** - JSON format
6. **Type hints** - All functions typed
7. **Comprehensive tests** - 80%+ coverage
8. **Docker optimization** - Multi-stage build, small image

DO NOT:

- Use sync functions where async is needed
- Skip error handling
- Forget type hints
- Use Pydantic v1 syntax (use v2)
- Make Docker image larger than necessary
- Skip tests

SPECIAL NOTES:

- The orchestrator `/agents/register` endpoint doesn't exist yet (will be added in Prompt 0.6)
 - For now, just log the registration attempt
 - Uncomment the actual registration code in Prompt 0.6
-

EXECUTE ALL TASKS. CREATE COMPLETE, WORKING FASTAPI APPLICATION. THIS PROVES WINDSURF CAN HANDLE PYTHON + FASTAPI GENERATION.